

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant: Charles BERNASCONI et al.

Title: NOTIFICATION OF
EMPLOYEES VIA PASS
CODE ACCESSED WEB
PAGES

Appl. No.: 09/641,866

Filing Date: 08/18/2000

Examiner: Kristine K. Rapillo

Art Unit: 3626

Confirmation Number: 7547

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

DECLARATION OF MR. ROLAND THOMPSON

Dear Sir:

I, the undersigned Roland Thompson, an American citizen with an office at 5 Great Valley Parkway, Malvern, Pennsylvania, USA, hereby declare and state that:

Background

1. I am a co-founder of the company Frontline Placement Technologies, Inc. (hereafter “Frontline”), and currently serve as a Managing Partner for the company. Frontline is the assignee for the present application.

2. Michael Blackstone and I conceived of the idea of a Web based substitute fulfillment system and founded Frontline Data, Inc., later renamed Frontline Placement Technologies, Inc., in 1998. We now have 65 employees, and provide substitute fulfillment services for over 1700 school districts in 47 states. We also provide shift fulfillment for manufacturing operations for a number of companies using Web fulfillment. I was the concept designer for the software for the automated substitute fulfillment system for Frontline Placement Technologies, Inc.

3. In 1996 I co-founded Thompson & Blackstone, Inc., a technical services and business company thinktank.

4. I was the co-founder and CEO of CF InFlight, LLP, which introduced Skycam (www.skycam.tv) to the sports broadcasting world. As the chief designer of the software for Skycam and chief executive, I brought Skycam to its present status, where it is featured on many of the worlds premier sports broadcasts including ESPN NFL broadcasts and several Super Bowls.

5. Prior to these ventures, I spent 10 years building an information technology services organization, Cone Software, focused on delivering IT services and software for stock and currency trading applications and developing mobile technology solutions for Fortune 500 companies. In that company, I designed and supervised the design of software for multiple platforms and for multiple applications.

6. I received a B.S. degree in computer science from West Chester University of Pennsylvania.

7. I am familiar with the claims and specification of the present patent application.

Meaning of the term “immediate response” and “immediately removing” in the Claim Element:

the one or more computers configured for automatically assigning the new open position only to one of the one or more preferred workers during a specified time period, in *immediate response* to receipt of an electronic selection of the new open position from one of the one or more preferred workers and *immediately removing* the position as an available for selection open position;

the one or more computers configured for assigning the new open position, after the expiration of the specified time period, to one of the qualified workers for which the new open position is made available for selection in *immediate response* to receipt of an electronic selection of the new open position from that qualified worker.

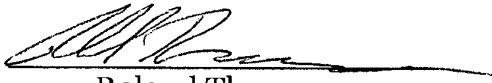
8. One of ordinary skill in the information technology art would understand that the use of the word “*immediate response*” in the above-recited claim element means that the system, comprising the one or more computers, starts a transaction to fulfill the position on receipt of the acceptance. The transaction may comprise, as one of ordinary skill would be aware, multiple different operations including one or more accesses to a database. No other worker acceptance subsequently received can jump ahead or override this transaction. In this respect, see the discussion of the term transaction in the “Sybase SQL Server Transact-SQL User’s Guide,” Release 11.0, last revised December 15, 1995, which is attached as an Exhibit to this Declaration. Note that this does not require that the posting on the respective web pages immediately be removed. But it does mean that it is no longer possible for another worker to accept the position. This would be clear to one of ordinary skill in the art.

9. I hereby declare that all statements made herein, unless otherwise indicated, are of my own knowledge and are true, and that all statements made on information and belief are believed to be true; and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under 18

087354-0108

U.S.C. § 1001, and that such willful false statements can jeopardize the validity of any patent issuing from the captioned application or claiming the benefit of its priority.

Dated: 7/21, 2009
Malvern, PA

Signed by: 
Roland Thompson
Managing Partner, Frontline Placement Technologies, Inc.

Sybase SQL Server™ Transact-SQL® User's Guide

Sybase SQL Server Release 11.0.x
Document ID: 32300-01-1100-02
Last Revised: December 15, 1995

Principal author: Server Publications Group

Document ID: 32300-01-1100

This publication pertains to Sybase SQL Server Release 11.0.x of the Sybase database management software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of that agreement.

Document Orders

To order additional documents, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the above fax number. All other international customers should contact their Sybase subsidiary or local distributor.

Upgrades are provided only at regularly scheduled software release dates.

Copyright © 1989–1995 by Sybase, Inc. All rights reserved.

No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without the prior written permission of Sybase, Inc.

Sybase Trademarks

APT-FORMS, Data Workbench, DBA Companion, Deft, GainExposure, Gain *Momentum*, Navigation Server, PowerBuilder, Powersoft, Replication Server, SA Companion, SQL Advantage, SQL Debug, SQL Monitor, SQL SMART, SQL Solutions, SQR, SYBASE, the Sybase logo, Transact-SQL, and VQL are registered trademarks of Sybase, Inc. Adaptable Windowing Environment, ADA Workbench, AnswerBase, Application Manager, APT-Build, APT-Edit, APT-Execute, APT-Library, APT-Translator, APT Workbench, Backup Server, Bit-Wise, Client-Library, Client/Server Architecture for the Online Enterprise, Client/Server for the Real World, Client Services, Configurator, Connection Manager, Database Analyzer, DBA Companion Application Manager, DBA Companion Resource Manager, DB-Library, Deft Analyst, Deft Designer, Deft Educational, Deft Professional, Deft Trial, Developers Workbench, DirectCONNECT, Easy SQR, Embedded SQL, EMS, Enterprise Builder, Enterprise Client/Server, Enterprise CONNECT, Enterprise Manager, Enterprise SQL Server Manager, Enterprise Work Architecture, Enterprise Work Designer, Enterprise Work Modeler, EWA, Gain Interplay, Gateway Manager, InfoMaker, Interactive Quality Accelerator, Intermedia Server, IQ Accelerator, Maintenance Express, MAP, MDI, MDI Access Server, MDI Database Gateway, MethodSet, Movedb, Navigation Server Manager, Net-Gateway, Net-Library, New Media Studio, OmniCONNECT, OmniSQL Access Module, OmniSQL Gateway, OmniSQL Server, OmniSQL Toolkit, Open Client, Open Client/Server, Open Client/Server Interfaces, Open Gateway, Open Server, Open Solutions, PC APT-Execute,

PC DB-Net, PC Net Library, Powersoft Portfolio, Replication Agent, Replication Driver, Replication Server Manager, Report-Execute, Report Workbench, Resource Manager, RW-DisplayLib, RW-Library, SAFE, SDF, Secure SQL Server, Secure SQL Toolset, SKILS, SQL Anywhere, SQL Code Checker, SQL Edit, SQL Edit/TPU, SQL Server, SQL Server/CFT, SQL Server/DBM, SQL Server Manager, SQL Server Monitor, SQL Station, SQL Toolset, SQR Developers Kit, SQR Execute, SQR Toolkit, SQR Workbench, Sybase Client/Server Interfaces, Sybase Gateways, Sybase Intermedia, Sybase Interplay, Sybase IQ, Sybase MPP, Sybase SQL Desktop, Sybase SQL Lifecycle, Sybase SQL Workgroup, Sybase Synergy Program, Sybase Virtual Server Architecture, Sybase User Workbench, SyBooks, System 10, System 11, the System XI logo, Tabular Data Stream, The Enterprise Client/Server Company, The Online Information Center, Warehouse WORKS, Watcom SQL, WebSights, WorkGroup SQL Server, XA-Library, and XA-Server are trademarks of Sybase, Inc.

All other company and product names used herein may be trademarks or registered trademarks of their respective companies.

Restricted Rights

Use, duplication, or disclosure by the government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

Sybase, Inc., 6475 Christie Avenue, Emeryville, CA 94608.

Table of Contents

Preface

Audience	xxiii
How to Use This Book	xxiii
Related Documents	xxiv
Conventions Used in This Manual	xxv
Formatting SQL Statements	xxv
SQL Syntax Conventions	xxvi
Case	xxvii
Obligatory Options {You Must Choose At Least One}	xxvii
Optional Options [You Don't Have to Choose Any]	xxvii
Ellipsis: Do It Again (and Again)... ..	xxvii
Expressions	xxviii
If You Need Help	xxviii

1. Introduction

Overview	1-1
Queries, Data Modification, and Commands	1-1
Tables, Columns, and Rows	1-2
The Relational Operations	1-3
Naming Conventions	1-3
SQL Data Characters	1-4
SQL Language Characters	1-4
Identifiers	1-5
Delimited Identifiers	1-6
Naming Conventions	1-6
Identifying Remote Servers	1-8
Transact-SQL Extensions	1-9
The <i>compute</i> Clause	1-9
Control-of-Flow Language	1-9
Stored Procedures	1-10
Triggers	1-10
Rules and Defaults	1-11
Error Handling and Set Options	1-11
Additional SQL Server Extensions to SQL	1-11
Compliance to Standards	1-12
FIPS Flagging	1-13
Chained Transactions and Isolation Levels	1-14

Delimited Identifiers	1-14
SQL Standard-Style Comments	1-14
Right Truncation of Character Strings	1-14
Permissions Required for <i>update</i> and <i>delete</i> Statements	1-15
Arithmetic Errors	1-15
Synonymous Keywords.....	1-16
Treatment of Nulls	1-16
How to Use Transact-SQL with the <i>isql</i>/Utility	1-16
Choosing a Password.....	1-17
Default Databases.....	1-17
Using the <i>pubs2</i> Sample Database	1-18
What Is in the Sample Database?.....	1-18

Part 1: Basic Concepts

2. Queries: Selecting Data from a Table

What Are Queries?	2-1
<i>select</i> Syntax.....	2-2
Choosing Columns in a Query	2-4
Choosing All Columns: <i>select *</i>	2-4
Choosing Some Columns	2-5
Rearranging the Order of Columns	2-6
Renaming Columns in Query Results.....	2-6
Quoted Strings in Column Headings	2-7
Character Strings in Query Results	2-8
Computed Values in the Select List	2-8
Arithmetic Operators.....	2-9
Arithmetic Operator Precedence	2-11
Selecting <i>text</i> and <i>image</i> Values.....	2-13
Using <i>readtext</i>	2-14
Select List Summary.....	2-15
Eliminating Duplicate Query Results with <i>distinct</i>.....	2-15
Specifying Tables: The <i>from</i> Clause	2-17
Selecting Rows: The <i>where</i> Clause	2-18
Comparison Operators	2-19
Ranges (<i>between</i> and <i>not between</i>)	2-21
Lists (<i>in</i> and <i>not in</i>)	2-22
Matching Character Strings: <i>like</i>	2-25
Using Wildcard Characters As Literal Characters.....	2-27
Square Brackets (Transact-SQL Extension).....	2-27

<i>escape</i> Clause (SQL Standard Compliant)	2-28
Interaction of Square Brackets and the <i>escape</i> Clause	2-29
Trailing Blanks and %	2-29
Using Wildcard Characters in Columns	2-29
Character Strings and Quotation Marks	2-30
“Unknown” Values: NULL	2-31
Connecting Conditions with Logical Operators	2-34
Logical Operator Precedence	2-34

3. Summarizing, Grouping, and Sorting Query Results

Summarizing Query Results Using Aggregate Functions	3-1
Aggregate Functions and Datatypes	3-3
Using <i>count</i> (*)	3-4
Using Aggregate Functions with <i>distinct</i>	3-5
Null Values and the Aggregate Functions	3-6
Organizing Query Results into Groups: The <i>group by</i> Clause	3-7
<i>group by</i> Syntax	3-8
Referencing Other Columns in Queries Using <i>group by</i>	3-10
Expressions and <i>group by</i>	3-13
Nesting Aggregates with <i>group by</i>	3-13
Null Values and <i>group by</i>	3-14
<i>where</i> Clause and <i>group by</i>	3-16
<i>group by</i> and <i>all</i>	3-17
Using Aggregates Without <i>group by</i>	3-18
Selecting Groups of Data: The <i>having</i> Clause	3-19
How the <i>having</i> , <i>group by</i> , and <i>where</i> Clauses Interact	3-21
Using <i>having</i> Without <i>group by</i>	3-23
Sorting Query Results: The <i>order by</i> Clause	3-24
<i>order by</i> and <i>group by</i>	3-26
Summarizing Groups of Data: The <i>compute</i> Clause	3-27
Row Aggregates and <i>compute</i>	3-30
Rules for <i>compute</i> Clauses	3-30
Specifying More Than One Column After <i>compute</i>	3-31
Using More Than One <i>compute</i> Clause	3-31
Applying an Aggregate to More Than One Column	3-32
Using Different Aggregates in the Same <i>compute</i> Clause	3-33
Grand Values: <i>compute</i> Without <i>by</i>	3-34
Combining Queries: The <i>union</i> Operator	3-35
Guidelines for <i>union</i> Queries	3-37
Using <i>union</i> with Other Transact-SQL Commands	3-38

4. Joins: Retrieving Data from Several Tables

What Are Joins?	4-1
Joins and the Relational Model	4-2
Joining Tables in Queries	4-3
The <i>from</i> Clause	4-4
The <i>where</i> Clause	4-4
How Joins Are Processed	4-6
Equijoins and Natural Joins	4-7
Joins with Additional Conditions	4-7
Joins Not Based on Equality	4-8
Self-Joins and Correlation Names	4-9
The Not-Equal Join	4-11
Not-Equal Joins and Subqueries	4-12
Joining More Than Two Tables	4-13
Outer Joins	4-15
Outer Join Restrictions	4-18
How Null Values Affect Joins	4-18
Determining Which Table Columns to Join	4-19

5. Subqueries: Using Queries Within Other Queries

What Are Subqueries?	5-1
Example of Using a Subquery	5-2
Subquery Syntax and General Rules	5-3
Subquery Restrictions	5-3
Qualifying Column Names	5-4
Subqueries with Correlation Names	5-4
Multiple Levels of Nesting	5-5
Subqueries in <i>update</i> , <i>delete</i> , and <i>insert</i> Statements	5-6
Subqueries in Conditional Statements	5-7
Using Subqueries in Place of an Expression	5-8
Types of Subqueries	5-9
Expression Subqueries	5-10
Using Scalar Aggregate Functions to Guarantee a Single Value	5-10
<i>group by</i> and <i>having</i> in Expression Subqueries	5-11
Using <i>distinct</i> with Expression Subqueries	5-12
Quantified Predicate Subqueries	5-12
Subqueries with <i>any</i> and <i>all</i>	5-13
> <i>all</i> Means Greater Than All Values	5-14
= <i>all</i> Means Equal to Every Value	5-14
> <i>any</i> Means Greater Than At Least One Value	5-15

=any Means Equal to Some Value	5-16
Subqueries Used with <i>in</i>	5-18
Subqueries Used with <i>not in</i>	5-20
Subqueries Using <i>not in</i> with NULL	5-21
Subqueries Used with <i>exists</i>	5-22
Subqueries Used with <i>not exists</i>	5-24
Finding Intersection and Difference with <i>exists</i>	5-25
Using Correlated Subqueries	5-26
Correlated Subqueries with Correlation Names	5-28
Correlated Subqueries with Comparison Operators	5-28
Correlated Subqueries in a <i>having</i> Clause	5-30

6. Using and Creating Datatypes

What Are Transact-SQL Datatypes?	6-1
Using System-Supplied Datatypes	6-2
Exact Numeric Types: Integers	6-3
Exact Numeric Types: Decimal Numbers	6-4
Approximate Numeric Datatypes	6-5
Character Datatypes	6-5
Binary Datatypes	6-7
Money Datatypes	6-8
Date and Time Datatypes	6-9
The <i>bit</i> Datatype	6-10
The <i>timestamp</i> Datatype	6-10
The <i>sysname</i> Datatype	6-11
Converting Between Datatypes	6-11
Mixed-Mode Arithmetic and Datatype Hierarchy	6-12
Working with <i>money</i> Datatypes	6-13
Determining Precision and Scale	6-13
Creating User-Defined Datatypes	6-14
Specifying Length, Precision, and Scale	6-15
Specifying Null Type	6-15
Associating Rules and Defaults with User-Defined Datatypes	6-16
Dropping a User-Defined Datatype	6-16
Getting Information About Datatypes	6-16

7. Creating Databases and Tables

What Are Databases and Tables?	7-1
Enforcing Data Integrity in Databases	7-2
Permissions Within Databases	7-3

Using and Creating Databases	7-3
Choosing a Database: <i>use</i>	7-4
Creating a User Database: <i>create database</i>	7-5
The <i>on</i> Clause	7-6
The <i>log on</i> Clause	7-7
The <i>for load</i> Option	7-8
Dropping Databases	7-8
Altering the Sizes of Databases	7-9
Creating Tables	7-10
Example of Creating a Table	7-10
Choosing Table Names	7-11
<i>create table</i> Syntax	7-12
Allowing Null Values	7-13
Using IDENTITY Columns	7-14
Creating IDENTITY Columns with User-Defined Datatypes	7-15
Referring to IDENTITY Columns with <i>syb_identity</i>	7-15
Generating Column Values	7-15
Using Temporary Tables	7-16
Creating Tables in Different Databases	7-17
Defining Integrity Constraints for Tables	7-18
Specifying Table-Level or Column-Level Constraints	7-19
Specifying Default Column Values	7-20
Specifying Unique and Primary Key Constraints	7-20
Specifying Referential Integrity Constraints	7-22
Specifying Check Constraints	7-23
How to Design and Create a Table	7-24
Make a Design Sketch	7-25
Create the User-Defined Datatypes	7-26
Choose the Columns That Accept Null Values	7-26
Define the Table	7-27
Creating New Tables from Query Results: <i>select into</i>	7-27
Selecting an IDENTITY Column	7-30
Adding a New IDENTITY Column with <i>select into</i>	7-31
Dropping Tables	7-31
Altering Existing Tables	7-32
Changing Table Structures: <i>alter table</i>	7-32
Renaming Tables and Other Objects	7-34
Effect of Renaming on Dependent Objects	7-35
Assigning Permissions to Users	7-35
Getting Information About Databases and Tables	7-36
Using <i>sp_help</i> on Database Objects	7-37

Using <i>sp_helpdb</i> on Databases	7-38
Using <i>sp_helpconstraint</i> on Tables	7-39
Using <i>sp_spaceused</i> on Tables.	7-39

8. Adding, Changing, and Deleting Data

What Choices Are Available to Modify Data?	8-1
Permissions	8-2
Referential Integrity	8-2
Transactions.	8-2
Using the Sample Database.	8-3
Datatype Entry Rules.	8-3
<i>char</i> , <i>nchar</i> , <i>varchar</i> , <i>nvarchar</i> , and <i>text</i>	8-4
<i>datetime</i> and <i>smalldatetime</i>	8-4
Entering Times	8-5
Entering Dates	8-6
Searching for Dates and Times.	8-8
<i>binary</i> , <i>varbinary</i> , and <i>image</i>	8-8
<i>money</i> and <i>smallmoney</i>	8-9
<i>float</i> , <i>real</i> , and <i>double precision</i>	8-9
<i>decimal</i> and <i>numeric</i>	8-10
<i>int</i> , <i>smallint</i> , and <i>tinyint</i>	8-11
<i>timestamp</i>	8-11
Adding New Data.	8-11
<i>insert</i> Syntax.	8-12
Adding New Rows with <i>values</i>	8-12
Inserting Data into Specific Columns	8-12
SQL Server-Generated Values for IDENTITY Columns.	8-13
Null Values, Defaults, IDENTITY Columns, and Errors	8-14
Explicitly Inserting Data into an IDENTITY Column.	8-15
Restricting Column Data: Rules.	8-15
Adding New Rows with <i>select</i>	8-16
Computed Columns	8-17
Inserting Data into Some Columns	8-18
Inserting Data from the Same Table	8-18
Changing Existing Data.	8-19
<i>update</i> Syntax	8-20
Using the <i>set</i> Clause with <i>update</i>	8-21
Using the <i>where</i> Clause with <i>update</i>	8-22
Using the <i>from</i> Clause with <i>update</i>	8-22

Changing <i>text</i> and <i>image</i> Data	8-23
Deleting Data	8-24
<i>delete</i> Syntax	8-24
Using the <i>where</i> Clause with <i>delete</i>	8-25
Using the <i>from</i> Clause with <i>delete</i>	8-25
Deleting All Rows from a Table	8-26
<i>truncate table</i> Syntax	8-26

9. Views: Limiting Access to Data

What Are Views?	9-1
Advantages of Views	9-2
Focus	9-2
Simpler Data Manipulation	9-2
Customization	9-2
Security	9-2
Logical Data Independence	9-4
View Examples	9-4
Creating Views	9-6
<i>create view</i> Syntax	9-6
Using the <i>select</i> Statement with <i>create view</i>	9-7
View Definition with Projection	9-8
View Definition with a Computed Column	9-8
View Definition with an Aggregate or Built-In Function	9-8
View Definition with a Join	9-9
Views Derived from Other Views	9-9
<i>distinct</i> Views	9-9
Views That Include IDENTITY Columns	9-10
Using the <i>with check option</i> Keyword with <i>create view</i>	9-11
Views Derived from Other Views	9-12
Limitations on Views Defined with Outer Joins	9-12
Retrieving Data Through Views	9-14
View Resolution	9-15
Redefining Views	9-15
Renaming Views	9-16
Altering or Dropping Underlying Objects	9-17
Modifying Data Through Views	9-17
Restrictions on Updating Views	9-19
Computed Columns in View Definition	9-19
<i>group by</i> or <i>compute</i> in View Definition	9-19
Null Values in Underlying Objects	9-20
Views Created <i>with check option</i>	9-21

Multitable Views	9-21
Views That Include IDENTITY Columns	9-22
Dropping Views	9-22
Using Views As Security Mechanisms	9-22
Getting Information About Views	9-23

Part 2: Advanced Topics

10. Using the Built-In Functions in Queries

System Functions	10-1
Examples of Using System Functions	10-6
<i>col_length</i>	10-6
<i>datalength</i>	10-6
<i>isnull</i>	10-7
<i>user_name</i>	10-7
String Functions	10-7
Examples of Using String Functions	10-11
<i>charindex</i> and <i>patindex</i>	10-11
<i>str</i>	10-12
<i>stuff</i>	10-13
<i>soundex</i> and <i>difference</i>	10-13
<i>substring</i>	10-14
Concatenation	10-15
Concatenation and the Empty String	10-16
Nested String Functions	10-16
Text Functions	10-17
Examples of Using Text Functions	10-18
Mathematical Functions	10-19
Examples of Using Mathematical Functions	10-22
Date Functions	10-23
Get Current Date: <i>getdate</i>	10-25
Find Date Parts As Numbers or Names	10-25
Calculate Intervals or Increment Dates	10-26
Add Date Interval: <i>dateadd</i>	10-27
Datatype Conversion Functions	10-28
Supported Conversions	10-28
Using the General-Purpose Conversion Function: <i>convert</i>	10-29
Conversion Rules	10-30
Converting Character Data to a Noncharacter Type	10-30
Converting from One Character Type to Another	10-31

Converting Numbers to a Character Type	10-31
Rounding During Conversion to or from Money Types	10-31
Converting Date and Time Information	10-32
Converting Between Numeric Types	10-32
Converting Binary-Like Data	10-32
Converting Hexadecimal Data	10-33
Converting <i>image</i> Data to <i>binary</i> or <i>varbinary</i>	10-33
Conversion Errors	10-34
Arithmetic Overflow and Divide-by-Zero Errors	10-34
Scale Errors	10-34
Domain Errors	10-35

11. Creating Indexes on Tables

What Are Indexes?	11-1
Comparing the Two Ways to Create Indexes	11-2
Guidelines for Using Indexes	11-2
Creating Indexes to Speed Up Data Retrieval.	11-3
<i>create index</i> Syntax	11-4
Indexing More Than One Column: Composite Indexes	11-5
Using the <i>unique</i> Option	11-6
Including IDENTITY Columns in Nonunique Indexes	11-6
Using the <i>fillfactor</i> and <i>max_rows_per_page</i> Options	11-7
<i>fillfactor</i>	11-7
<i>max_rows_per_page</i>	11-7
Using Clustered or Nonclustered Indexes	11-8
Specifying Index Options	11-10
Using the <i>ignore_dup_key</i> Option	11-10
Using the <i>ignore_dup_row</i> and <i>allow_dup_row</i> Options	11-11
Using the <i>sorted_data</i> Option	11-12
Using the <i>on segment_name</i> Option	11-12
Dropping Indexes	11-13
Determining What Indexes Exist on a Table	11-13
Updating Statistics About Indexes.	11-13

12. Defining Defaults and Rules for Data

What Are Defaults and Rules?	12-1
Comparing Defaults and Rules with Integrity Constraints	12-2
Creating Defaults	12-2
<i>create default</i> Syntax	12-3

Binding Defaults	12-4
Unbinding Defaults	12-7
Dropping Defaults	12-8
How Defaults Affect Null Values	12-8
Creating Rules	12-9
<i>create rule</i> Syntax	12-9
Binding Rules	12-10
Rules Bound to Columns	12-11
Rules Bound to User-Defined Datatypes	12-11
Precedence of Rules	12-11
Unbinding Rules	12-12
Dropping Rules	12-13
Getting Information About Defaults and Rules	12-14

13. Using Batches and Control-of-Flow Language

What Are Batches and Control-of-Flow Language?	13-1
Rules Associated with Batches	13-2
Examples of Using Batches	13-3
Batches Submitted As Files	13-5
Using Control-of-Flow Language	13-6
<i>if...else</i>	13-7
<i>begin...end</i>	13-8
<i>while</i> and <i>break...continue</i>	13-9
<i>declare</i> and Local Variables	13-12
Variables and Null Values	13-14
<i>declare</i> and Global Variables	13-15
<i>goto</i>	13-18
<i>return</i>	13-19
<i>print</i>	13-20
<i>raiserror</i>	13-22
User-Defined Messages for <i>print</i> and <i>raiserror</i>	13-23
<i>waitfor</i>	13-24
Comments	13-25

14. Using Stored Procedures

What Are Stored Procedures?	14-1
Examples of Creating and Using Stored Procedures	14-2
Stored Procedures and Permissions	14-4
Stored Procedures and Performance	14-4

Creating and Executing Stored Procedures	14-4
Parameters	14-5
Default Parameters	14-7
NULL As Default Parameter	14-9
Wildcard Characters in the Default Parameter	14-9
Using More Than One Parameter	14-9
Procedure Groups	14-11
<i>with recompile</i> in <i>create procedure</i>	14-11
<i>with recompile</i> in <i>execute</i>	14-11
Nesting Procedures Within Procedures	14-12
Using Temporary Tables in Stored Procedures	14-12
Executing Procedures Remotely	14-13
Returning Information from Stored Procedures	14-14
Return Status	14-15
Reserved Return Status Values	14-15
User-Generated Return Values	14-16
Checking Roles in Procedures	14-16
Return Parameters	14-17
Passing Values in Parameters	14-21
The <i>output</i> Keyword	14-21
Rules Associated with Stored Procedures	14-22
Qualifying Names Inside Procedures	14-23
Dropping Stored Procedures	14-23
Renaming Stored Procedures	14-24
Renaming Objects Referenced by Procedures	14-24
Using Stored Procedures As Security Mechanisms	14-24
System Procedures	14-25
Security Administration	14-26
Remote Servers	14-26
Data Definition and Database Objects	14-26
User-Defined Messages	14-27
System Administration	14-27
Getting Information About Stored Procedures	14-28
<i>sp_help</i>	14-28
<i>sp_helptext</i>	14-28
<i>sp_depends</i>	14-29

15. Triggers: Enforcing Referential Integrity

What Are Triggers?	15-1
Comparing Triggers with Integrity Constraints	15-2

Creating Triggers	15-3
<i>create trigger</i> Syntax	15-3
SQL Statements Not Allowed in Triggers	15-4
Dropping Triggers	15-5
Using Triggers to Maintain Referential Integrity	15-5
How Triggers Work	15-6
Testing Data Modifications Against the Trigger Test Tables	15-6
An Insert Trigger Example	15-8
A Delete Trigger Example	15-9
Update Trigger Examples	15-11
Updating a Foreign Key	15-12
Multirow Considerations	15-13
A Conditional Insert Trigger	15-16
Rolling Back Triggers	15-17
Nesting Triggers	15-19
Trigger Self-Recursion	15-20
Rules Associated with Triggers	15-22
Triggers and Permissions	15-22
Trigger Restrictions	15-22
Implicit and Explicit Null Values	15-23
Triggers and Performance	15-24
<i>set</i> Commands in Triggers	15-24
Renaming and Triggers	15-24
Getting Information About Triggers	15-25
<i>sp_help</i>	15-25
<i>sp_helptext</i>	15-25
<i>sp_depends</i>	15-26

16. Cursors: Accessing Data Row by Row

What Are Cursors?	16-1
How SQL Server Processes Cursors	16-2
Declaring Cursors	16-3
<i>declare cursor</i> Syntax	16-3
Cursor Scope	16-4
Cursor Scans and the Cursor Result Set	16-5
Making Cursors Updatable	16-6
Opening Cursors	16-7
Fetching Data Rows Using Cursors	16-8
<i>fetch</i> Syntax	16-8
Checking the Cursor Status	16-9

Checking the Number of Rows Fetched	16-10
Getting Multiple Rows with Each <i>fetch</i>	16-10
Updating and Deleting Rows Using Cursors	16-11
Deleting Cursor Result Set Rows	16-11
Updating Cursor Result Set Rows	16-12
Closing and Deallocating Cursors	16-13
An Example Using a Cursor	16-14
Using Cursors in Stored Procedures	16-16
Cursors and Locking	16-18
Getting Information About Cursors	16-19

17. Transactions: Maintaining Data Consistency and Recovery

What Are Transactions?	17-1
Transactions and Consistency	17-2
Transactions and Recovery	17-2
Using Transactions	17-2
Allowing Data Definition Commands in Transactions	17-3
Beginning and Committing Transactions	17-4
Rolling Back and Saving Transactions	17-5
Checking the State of Transactions	17-6
Nested Transactions	17-8
Example of a User-Defined Transaction	17-9
Selecting Transaction Mode and Isolation Level	17-10
Choosing a Transaction Mode	17-11
Choosing an Isolation Level	17-12
Changing the Isolation Level for a Query	17-14
Cursors and Isolation Levels	17-15
Stored Procedures and Isolation Levels	17-16
Triggers and Isolation Levels	17-16
Using Transactions in Stored Procedures and Triggers	17-17
Transaction Modes and Stored Procedures	17-19
Setting Transaction Modes for Stored Procedures	17-20
Using Cursors in Transactions	17-21
Backup and Recovery of Transactions	17-21

Glossary

Index

17 Transactions: Maintaining Data Consistency and Recovery

Transactions provide a way to group Transact-SQL statements so that they are treated as a unit. Either all statements in the group are executed or no statements are executed.

This chapter discusses:

- An overview of transactions
- How to use group statements in a transaction
- How to define transaction modes and isolation levels
- How stored procedures and triggers work with transactions
- How cursors work with transactions
- Backup and recovery of transactions

What Are Transactions?

A transaction is a mechanism for ensuring that a set of one or more SQL statements is treated as a single unit of work. SQL Server automatically manages all data modification commands, including single-step change requests, as transactions. By default, each **insert**, **update**, and **delete** statement is considered a single transaction.

You can group a set of SQL statements into a user-defined transaction with the **begin transaction**, **commit transaction**, and **rollback transaction** commands. **begin transaction** marks the beginning of a transaction block. All subsequent statements, up to a **rollback transaction** or a matching **commit transaction**, are included as part of the transaction.

Transactions allow SQL Server to guarantee:

- Consistency – Simultaneous queries and change requests cannot collide with each other, and users never see or operate on data that is part way through a change.
- Recovery – In case of system failure, database recovery is complete and automatic.

To support SQL standards-compliant transactions, SQL Server provides options that allow you to select the mode and isolation level for your transactions. Applications that require SQL standards-compliant transactions should set those options at the beginning of

every session. Transaction modes and isolation levels are described later in this chapter.

Transactions and Consistency

In a multiuser environment, SQL Server must prevent simultaneous queries and data modification requests from interfering with each other. This is important because if the data being processed by a query could be changed by another user's update while the query runs, the results of the query would be ambiguous.

SQL Server automatically sets the appropriate level of locking for each transaction. You can make shared locks more restrictive on a query-by-query basis by including the **holdlock** keyword in a **select** statement.

User-defined transactions allow users to instruct SQL Server to process any number of SQL statements as a single unit. They are discussed in a later section.

Transactions and Recovery

A transaction is both a unit of work and a unit of recovery. The fact that SQL Server handles single-step change requests as transactions means that the database can be recovered completely in case of failures.

SQL Server's recovery time is measured in seconds and minutes. You can specify the maximum acceptable recovery time.

The SQL commands related to recovery and backup are discussed in "Backup and Recovery of Transactions" on page 17-21.

Using Transactions

begin transaction and **commit transaction** tell SQL Server to process any number of single commands as a single unit. **rollback transaction** undoes the transaction, either back to its beginning, or back to a savepoint. You define a **savepoint** inside a transaction with the **save transaction** command.

User-defined transactions give you control over transaction management. They also improve performance, since system overhead is incurred once per transaction, rather than once for each individual command.

► **Note**

Grouping large numbers of Transact-SQL commands into one long-running transaction may affect recovery time. If SQL Server fails before the transaction commits, recovery is longer, because SQL Server must undo the transaction.

Any user can define a transaction. No permission is required for any of the transaction commands.

The following sections discuss general transaction topics and transaction commands, with examples. For more information about transactions, see the *SQL Server Reference Manual*.

Allowing Data Definition Commands in Transactions

You can use certain data definition language commands in transactions by setting `sp_dboption`'s `ddl in tran` option to true. If `ddl in tran` is true in a particular database, you can issue commands such as **create table**, **grant**, and **alter table** inside transactions in that database. If `ddl in tran` is true in the *model* database, you can issue the commands inside transactions in all databases created after `ddl in tran` was set to true in *model*.

◆ **WARNING!**

The only scenario in which using data definition language commands inside transactions is justified is in create schema. Data definition language commands hold locks on system tables such as *sysobjects*. If you use data definition language commands inside transactions, keep the transactions short.

In particular, avoid using any data definition language commands on *tempdb* within transactions, lest your system grind to a halt. Always leave `ddl in tran` set to false in *tempdb*.

To set `ddl in tran` to true, type:

```
sp_dboption mydb,"ddl in tran", true
```

The first parameter specifies the name of the database in which to set the option. You must be using the *master* database to execute `sp_dboption`. Any user can execute `sp_dboption` with no parameters to

display the current option settings. To set options, however, you must be either a System Administrator or the Database Owner.

The following commands are allowed inside a user-defined transaction only if the **ddl in tran** option to **sp_dboption** is set to true:

Table 17-1: DDL commands allowed in transactions

alter table (clauses other than partition and unpartition are allowed)	create default create index create procedure create rule create schema create table create trigger create view	drop default drop index drop procedure drop rule drop table drop trigger drop view	grant revoke
--	---	---	-------------------------------

System procedures that change the *master* database or create temporary tables cannot be used inside user-defined transactions.

Never use the following commands inside a user-defined transaction:

Table 17-2: DDL commands not allowed in transactions

alter database alter table...partition alter table...unpartition create database	disk init dump database dump transaction drop database	load transaction load database reconfigure	select into update statistics truncate table
---	---	---	---

You can check the current setting of **ddl in tran** with **sp_helpdb**.

Beginning and Committing Transactions

The **begin transaction** and **commit transaction** commands can enclose any number of SQL statements and stored procedures. The syntax for both statements is:

```
begin {transaction | tran} [transaction_name]  
commit {transaction | tran | work} [transaction_name]
```

transaction_name is the name assigned to the transaction. It must conform to the rules for identifiers.

The keywords **transaction**, **tran**, and **work** (in **commit transaction**) are synonymous; you can use one in the place of the others. However,

transaction and **tran** are Transact-SQL extensions; only **work** is SQL standards-compliant.

Here is a skeletal example:

```
begin tran
    statement
    procedure
    statement
commit tran
```

commit transaction does not affect SQL Server if a transaction is not currently active.

Rolling Back and Saving Transactions

If a transaction must be canceled before it is committed—either because of some failure or because of a change by the user—all of its completed statements or procedures must be undone.

You can cancel or roll back a transaction with the **rollback transaction** command at any time before the **commit transaction** command has been given. Using savepoints, you can cancel either an entire transaction or part of it. However, you cannot cancel a transaction after it has been committed.

The syntax of the **rollback transaction** command is:

```
rollback {transaction | tran | work}
        [transaction_name | savepoint_name]
```

A **savepoint** is a marker that the user puts inside a transaction to indicate a point to which it can be rolled back.

Savepoints are inserted by putting a **save transaction** command within the transaction. The syntax is:

```
save {transaction | tran} savepoint_name
```

The savepoint name must conform to the rules for identifiers.

If no *savepoint_name* or *transaction_name* is given with the **rollback transaction** command, the transaction is rolled back to the first **begin transaction** in a batch.

Here is how you can use the **save transaction** and **rollback transaction** commands:

```

begin tran transaction_name
    statement
    statement
    procedure
save tran savepoint_name
    statement
rollback tran savepoint_name
    statement
    statement
rollback tran

```

The first **rollback transaction** command rolls the transaction back to the savepoint inside the transaction. The second **rollback transaction** rolls the transaction back to its beginning. If a transaction is rolled back to a savepoint, it must still proceed to completion or else be canceled altogether.

Until you issue a **commit transaction**, SQL Server considers all subsequent statements to be part of the transaction, unless it encounters another **begin transaction** statement. At that point, SQL Server considers all subsequent statements to be part of this new nested transaction. Nested transactions are described in the next section.

rollback transaction or **save transaction** does not affect SQL Server and does not return an error message if a transaction is not currently active.

Checking the State of Transactions

The global variable @@*transtate* keeps track of the current state of a transaction. SQL Server determines what state to return by keeping track of any transaction changes after a statement executes. @@*transtate* may contain the following values:

Table 17-3: @@transtate values

Value	Meaning
0	Transaction in progress. An explicit or implicit transaction is in effect; the previous statement executed successfully.
1	Transaction succeeded. The transaction completed and committed its changes.
2	Statement aborted. The previous statement was aborted; no effect on the transaction.
3	Transaction aborted. The transaction aborted and rolled back any changes.

In a transaction, you can use `@@transtate` after a statement (such as an `insert`) to determine whether it was successful or aborted, and to determine its effect on the transaction. The following example checks `@@transtate` during a transaction (after a successful `insert`) and after the transaction commits:

```
begin transaction

insert into publishers (pub_id) values ('9999')
(1 row affected)
select @@transtate
-----
          0

(1 row affected)
commit transaction

select @@transtate
-----
          1

(1 row affected)
```

This next example checks `@@transtate` after an unsuccessful `insert` (due to a rule violation) and after the transaction rolls back:

```
begin transaction

insert into publishers (pub_id) values ('7777')
Msg 552, Level 16, State 1:
A column insert or update conflicts with a rule
bound to the column. The command is aborted. The
conflict occurred in database 'pubs2', table
'publishers', rule 'pub_idrule', column 'pub_id'.

select @@transtate
-----
          2

(1 row affected)
rollback transaction
select @@transtate
```

```
-----
3
```

```
(1 row affected)
```

Unlike `@@error`, however, SQL Server does not clear `@@transtate` after every statement. It changes `@@transtate` only in response to an action taken by a transaction.

Nested Transactions

You can nest transactions within other transactions. When you nest **begin transaction** and **commit transaction** statements, the outermost pair actually begin and commit the transaction. The inner pairs just keep track of the nesting level. SQL Server does not commit the transaction until the **commit transaction** that matches the outermost **begin transaction** is issued.

SQL Server provides a global variable, `@@trancount`, that keeps track of the current nesting level for transactions. An initial implicit or explicit **begin transaction** sets `@@trancount` to 1. Each subsequent **begin transaction** increments `@@trancount`, and a **commit transaction** decrements it. Firing a trigger also increments `@@trancount`, and the transaction begins with the statement that causes the trigger to fire. Nested transactions are not committed until `@@trancount` equals 0.

For example, the following nested groups of statements are not committed by SQL Server until the final **commit transaction**:

```
begin tran
  select @@trancount
  /* @@trancount = 1 */

  begin tran
    select @@trancount
    /* @@trancount = 2 */

    begin tran
      select @@trancount
      /* @@trancount = 3 */
      commit tran

    commit tran

  commit tran

commit tran
```

```
select @@trancount
/* @@ trancount = 0 */
```

When you nest a **rollback transaction** statement without including a transaction or savepoint name, it always rolls back to the outermost **begin transaction** statement and cancels the transaction.

Example of a User-Defined Transaction

This example shows how a user-defined transaction might be specified:

```
begin transaction royalty_change

/* A user sets out to change the royalty split */
/* for the two authors of The Gourmet Microwave. */
/* Since the database would be inconsistent */
/* between the two updates, they must be grouped */
/* into a transaction. */

update titleauthor
set royaltypers = 65
from titleauthor, titles
where royaltypers = 75
and titleauthor.title_id = titles.title_id
and title = "The Gourmet Microwave"

update titleauthor
set royaltypers = 35
from titleauthor, titles
where royaltypers = 25
and titleauthor.title_id = titles.title_id
and title = "The Gourmet Microwave"

save transaction percent_changed

/* After updating the royaltypers entries for */
/* the two authors, the user inserts the */
/* savepoint "percent_changed," and then checks */
/* to see how a 10 percent increase in the */
/* price would affect the authors' royalty */
/* earnings. */
```

```
update titles
set price = price * 1.1
where title = "The Gourmet Microwave"

select (price * royalty * total_sales) * royaltypcr
from titles, titleauthor, roysched
where title = "The Gourmet Microwave"
and titles.title_id = titleauthor.title_id
and titles.title_id = roysched.title_id

rollback transaction percent_changed

/* The transaction rolls back to the savepoint */
/* with the rollback transaction command. */
/* Without a savepoint, it would roll back to */
/* the begin transaction. */

commit transaction
```

Selecting Transaction Mode and Isolation Level

SQL Server provides two options you can set to support SQL standard-compliant transactions. These options define the transaction mode and transaction isolation level. You should set these options at the beginning of every session that requires SQL standards-compliant transactions.

SQL Server supports the following transaction modes:

- The default mode, called **unchained** or Transact-SQL mode, requires explicit **begin transaction** statements paired with **commit transaction** or **rollback transaction** statements to complete the transaction.
- The SQL standards-compatible mode, called **chained** mode, implicitly begins a transaction before any data retrieval or modification statement. These statements include: **delete**, **insert**, **open**, **fetch**, **select**, and **update**. You must still explicitly end the transaction with **commit transaction** or **rollback transaction**.

You can set either mode using the **chained** option of the **set** command. However, you should not mix these transaction modes in your applications. The behavior of stored procedures and triggers can

vary depending on the mode, and you may require special action to run a procedure in one mode that was created in the other.

SQL Server supports the following transaction isolation levels:

- Level 0 – SQL Server ensures that data written by one transaction represents the actual data. This level prevents other transactions from writing over the same data until the transaction commits. The other transactions can still read the uncommitted data.
- Level 1 – SQL Server ensures that data read by one transaction represents the actual data, not the data in the process of another uncommitted transaction. This is the default isolation level supported by SQL Server.
- Level 3 – SQL Server ensures that data read by one transaction is valid until the end of that transaction. It supports this level through the **holdlock** keyword of the **select** statement which applies a read-lock on the specified data.

You can set the isolation level for your session using the **transaction isolation level** option of the **set** command. You can enforce the isolation level for just a query as opposed to using the **at isolation** clause of the **select** statement.

The following sections describe these options in more detail.

Choosing a Transaction Mode

The SQL standards require every SQL data-retrieval and data-modification statement to occur inside of a transaction. A transaction automatically starts with the first data-retrieval or data-modification statement after the start of a session or after the previous transaction commits or aborts. This is the chained transaction mode.

You can set this mode for your current session by turning on the **chained** option of the **set** statement. For example:

```
set chained on
```

However, you cannot execute the **set chained** command within a transaction. To return to the unchained transaction mode, set the **chained** option **off**. The default transaction mode is unchained.

In the chained transaction mode, SQL Server implicitly executes a **begin transaction** statement just before the following data retrieval or modification statements: **delete**, **insert**, **open**, **fetch**, **select**, and **update**. For example, the following group of statements produce different results depending on which mode you use:


```
insert into publishers
  values ('9999', null, null, null)
begin transaction
delete from publishers where pub_id = '9999'
rollback transaction
```

In unchained mode, the **rollback** affects only the **delete** statement, so *publishers* still contains the inserted row. In chained mode, the **insert** statement implicitly begins a transaction, and the rollback affects all statements up to the beginning of that transaction, including the **insert**.

Although chained mode implicitly begins transactions with data retrieval or modification statements, you can nest transactions only by explicitly using **begin transaction** statements. Once the first transaction implicitly begins, further data retrieval or modification statements no longer begin transactions until after the first transaction commits or aborts. For example, in the following query, the first **commit transaction** commits all changes in chained mode; the second commit is unnecessary:

```
insert into publishers
  values ('9999', null, null, null)
insert into publishers
  values ('9997', null, null, null)
commit transaction
commit transaction
```

► **Note**

In chained mode, a data retrieval or modification statement begins a transaction whether or not it executes successfully. Even a **select** that does not access a table begins a transaction.

You can check the global variable `@@tranchained` to determine SQL Server's current transaction mode. **select @@tranchained** returns a 0 for unchained mode or a 1 for chained mode.

Choosing an Isolation Level

The SQL92 standard defines four levels of isolation for transactions. Each isolation level specifies the kinds of actions that are not permitted while concurrent transactions are executing. Higher levels include the restrictions imposed by the lower levels:

- Level 0 prevents other transactions from changing data that has already been modified (through an **insert**, **delete**, **update**, and so on) by an uncommitted transaction. The other transactions are blocked from modifying that data until the transaction commits. However, other transactions can still read the uncommitted data, which results in **dirty reads**.
- Level 1 prevents dirty reads. Such reads occur when one transaction modifies a row, and then a second transaction reads that row before the first transaction commits the change. If the first transaction rolls back the change, the information read by the second transaction becomes invalid.
- Level 2 prevents **nonrepeatable reads**. Such reads occur when one transaction reads a row and a second transaction modifies that row. If the second transaction commits its change, subsequent reads by the first transaction yield different results than the original read.
- Level 3 prevents **phantoms**. Phantoms occur when one transaction reads a set of rows that satisfy a search condition, and then a second transaction modifies the data (through an **insert**, **delete**, **update**, and so on). If the first transaction repeats the read with the same search conditions, it obtains a different set of rows.

By default, SQL Server's transaction isolation level is 1. The SQL92 standard requires that level 3 be the default isolation for all transactions. This prevents dirty reads, nonrepeatable reads, and phantoms. To enforce this default level of isolation, Transact-SQL provides the **transaction isolation level 3** option of the **set** statement. This option instructs SQL Server to automatically apply a **holdlock** to all **select** operations in a transaction. For example:

```
set transaction isolation level 3
```

Applications that use **transaction isolation level 3** should set that isolation level at the beginning of each session. However, setting **transaction isolation level 3** causes SQL Server to hold any read-locks for the duration of the transaction. If you also use the chained transaction mode, that isolation level remains in effect for any data retrieval or modification statement that implicitly begins a transaction. In both cases, this can lead to concurrency problems for some applications, since more locks may be held for longer periods of time.

To return your session to the SQL Server default isolation level:

```
set transaction isolation level 1
```

Applications that are not impacted by dirty reads may see better concurrency and reduced deadlocks when accessing the same data

by setting **transaction isolation level 0** at the beginning of each session. An example is an application that finds the momentary average balance for all savings accounts stored in a table. Since it requires only a snapshot of the current average balance, which probably changes frequently in an active table, the application should query the table using isolation level 0. Other applications that require data consistency, such as deposits and withdrawals to specific accounts in the table, should avoid using isolation level 0.

Queries executing at isolation level 0 do not acquire any read locks for their scans, so they do not block other transactions from writing to the same data, and vice versa. However, even if you set your isolation level to 0, utilities (like **dbcc**) and data modification statements (like **update**) still acquire read locks for their scans, because they must maintain the database integrity by ensuring that the correct data has been read before modifying it.

The global variable `@@isolation` contains the current isolation level of your Transact-SQL session. Querying `@@isolation` returns the value of the active level (0, 1, or 3). For example:

```
select @@isolation
-----
          1

(1 row affected)
```

For more information about isolation levels and locking, see the *Performance and Tuning Guide*.

Changing the Isolation Level for a Query

You can change the isolation level for a query by using the **at isolation** clause with the **select** or **readtext** statements. The **read uncommitted**, **read committed**, and **serializable** options of **at isolation** represent each isolation level as defined below:

at isolation Option	Isolation Level
read uncommitted	0
read committed	1
serializable	3

For example, the following two statements query the same table at isolation levels 0 and 3, respectively:

```
select *
from titles
at isolation read uncommitted

select *
from titles
at isolation serializable
```

The **at isolation** clause is valid only for single **select** and **readtext** queries or in the **declare cursor** statement. SQL Server returns a syntax error if you use **at isolation** as follows:

- With a query using the **into** clause
- Within a subquery
- With a query in the **create view** statement
- With a query in the **insert** statement
- With a query using the **for browse** clause

If there is a **union** operator in the query, you must specify the **at isolation** clause after the last **select**.

The SQL92 standard defines **read uncommitted**, **read committed**, and **serializable** as options for **at isolation** (and **set transaction isolation level** as well). A Transact-SQL extension also allows you to specify 0, 1, or 3 for **at isolation**. To simplify the discussion of isolation levels, the **at isolation** examples in this manual do not use this extension.

You can also enforce isolation level 3 using the **holdlock** keyword of the **select** statement. However, you cannot specify **holdlock**, **noholdlock**, or **shared** in a query that also specifies **at isolation read uncommitted**. When you use different ways to set an isolation level, the **holdlock** keyword takes precedence over the **at isolation** clause (except for isolation level 0), and **at isolation** takes precedence over the session level defined by **set transaction isolation level**.

Cursors and Isolation Levels

You can use the **select** statement's **at isolation** clause to change the isolation level with a cursor. For example:

```
declare commit_crsr cursor
for select *
from titles
at isolation read committed
```

This statement makes the cursor operate at isolation level 1, regardless of the isolation level of the transaction or session. If you declare a cursor at isolation level 0 (**read uncommitted**), SQL Server also

defines the cursor as read-only. You cannot specify the **for update** clause along with **at isolation read uncommitted** in a **declare cursor** statement.

SQL Server decides a cursor's isolation level when you open it, not when it is declared. Once you open the cursor, SQL Server determines its isolation level based on the following:

- If the cursor was declared with the **at isolation** clause, that isolation level overrides the transaction isolation level in which it is opened.
- If the cursor was **not** declared with **at isolation**, the cursor uses the isolation level in which it is opened. If you close the cursor and reopen it later, the cursor acquires the current isolation level of the transaction.

The caveat to the last point is that certain types of cursors (language and client) declared in a transaction with isolation level 1 or 3 cannot be opened in a transaction with isolation level 0. For more information about this restriction and about the different types of cursors, see the *SQL Server Reference Manual*.

Stored Procedures and Isolation Levels

The Sybase system-stored procedures always operate at isolation level 1, regardless of the transaction or session isolation level. User stored procedures operate at the isolation level of the transaction that executes it. If the isolation level changes within a stored procedure, the new isolation level remains in effect only during the execution of the stored procedure.

Triggers and Isolation Levels

Since triggers are fired by data modification statements (like **insert**), all triggers execute at either the transaction's isolation level or isolation level 1, whichever is higher. So, if a trigger fires in a transaction at level 0, SQL Server sets the trigger's isolation level to 1 before executing its first statement.

Using Transactions in Stored Procedures and Triggers

You can use transactions in stored procedures and triggers just as with statement batches. If a transaction in a batch or stored procedure invokes another stored procedure or trigger containing a transaction, that second transaction is nested into the first one.

The first explicit or implicit (using chained mode) **begin transaction** starts the transaction in the batch, stored procedure, or trigger. Each subsequent **begin transaction** increments the nesting level. Each subsequent **commit transaction** decrements the nesting level until it reaches 0. SQL Server then commits the entire transaction. A **rollback transaction** aborts the entire transaction up to the first **begin transaction** regardless of the nesting level or the number of stored procedures and triggers it spans.

In stored procedures and triggers, the number of **begin transaction** statements must match the number of **commit transaction** statements. This also applies to stored procedures that use chained mode. The first statement that implicitly begins a transaction must also have a matching **commit transaction**.

The following illustration demonstrates what can happen when you nest transaction statements within stored procedures:

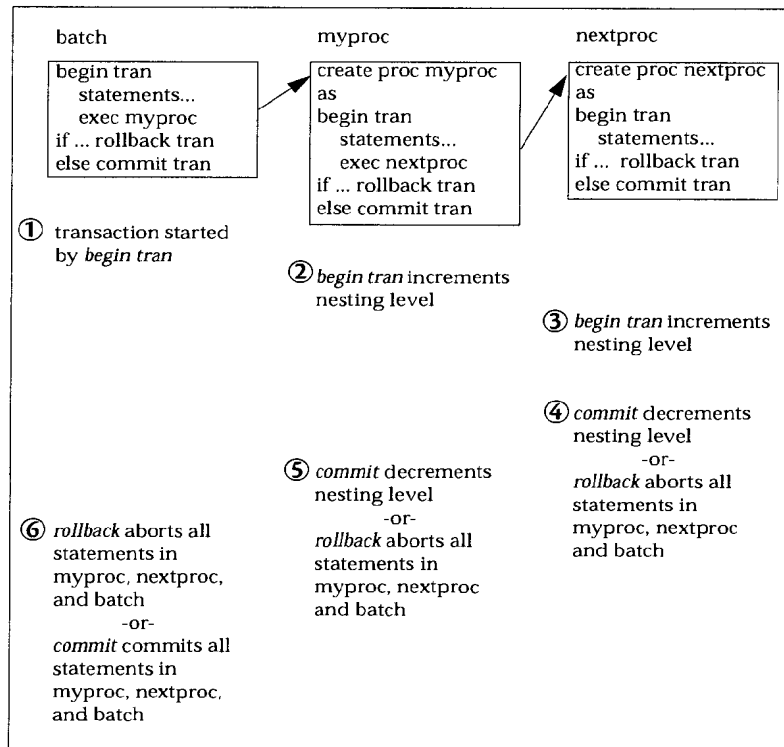


Figure 17-1: Nesting transaction statements

rollback transaction statements in stored procedures do not affect subsequent statements in the procedure or batch that originally called the procedure. SQL Server executes subsequent statements in the stored procedure or batch. However, **rollback transaction** statements in triggers do abort the batch so that subsequent statements are not executed.

For example, the following batch calls the stored procedure *myproc* which includes a **rollback transaction** statement:

```
begin tran
update titles set ...
insert into titles ...
execute myproc
delete titles where ...
```

The **update** and **insert** statements are rolled back and the transaction is aborted. SQL Server continues the batch and executes the **delete** statement. However, if there is an **insert** trigger on a table that includes a **rollback transaction**, the entire batch is aborted and the **delete** is not executed. For example:

```
begin tran
update authors set ...
insert into authors ...
delete authors where ...
```

Different transaction modes or isolation levels for stored procedures have certain requirements, which are described in the next section. Triggers are not affected by the current transaction mode since they are always called as part of a data modification statement.

Transaction Modes and Stored Procedures

Stored procedures written to use the unchained transaction mode may be incompatible with other transactions using chained mode, and vice versa. For example, following is a valid stored procedure using chained transaction mode:

```
create proc myproc
as
insert into publishers
values ('9999', null, null, null)
commit work
```

A program using unchained transaction mode would fail if it called this procedure because the **commit** does not have a corresponding **begin**. You may encounter other problems:

- Applications that start a transaction using chained mode may create impossibly long transactions, or may hold data locks for the entire length of their session. This behavior degrades SQL Server performance.
- Applications may nest transactions at unexpected times. This can produce different results depending on the transaction mode.

As a rule, applications using one transaction mode should call stored procedures written to use that same mode. The exceptions to that

rule are Sybase system-stored procedures (not including **sp_procxmode**, described below), which can be invoked by sessions using any transaction mode. If no transaction is active when you execute a system-stored procedure, SQL Server turns off chained mode for the duration of the procedure. Before returning, it resets the mode its original setting.

SQL Server tags all procedures with the transaction mode ("chained" or "unchained") of the session in which they are created. This helps avoid problems associated with transactions using one mode invoking other transactions using the other mode. A stored procedure tagged as "chained" is not executable in sessions using unchained transaction mode, and vice versa.

◆ **WARNING!**

When using transaction modes, be aware of the effects each setting can have on your applications.

Setting Transaction Modes for Stored Procedures

You can use the **sp_procxmode** system stored procedure to change the tag value associated with a stored procedure. SQL Server also provides a third tag, "anymode", which you can use with **sp_procxmode** to indicate stored procedures that can run under either transaction mode. For example:

```
sp_procxmode byroyalty, "anymode"
```

Use **sp_procxmode** without any parameter values to get the transaction modes for all stored procedures in the current database:

```
sp_procxmode
procedure name          transaction mode
-----
byroyalty              Unchained
discount_proc         Unchained
insert_sales_proc     Unchained
insert_salesdetail_proc Unchained
storeid_proc          Unchained
storename_proc        Unchained
title_proc            Unchained
titleid_proc          Unchained
```

```
(8 rows affected, return status = 0)
```

You can use **sp_procxmode** only in unchained transaction mode.

To change a procedure's transaction mode, you must be a System Administrator, the Database Owner, or the owner of the procedure.

Using Cursors in Transactions

By default, SQL Server does not change a cursor's state (open or closed) when a transaction ends through a commit or roll back. The SQL standards, however, associate an open cursor with its active transaction. Committing or rolling back that transaction automatically closes any open cursors associated with it.

To enforce this SQL standards-compliant behavior, SQL Server provides the **close on endtran** option of the **set** command. In addition, if you set chained mode on, SQL Server starts a transaction when you open a cursor, and it closes that cursor when the transaction is committed or rolled back.

For example, the following sequence of statements produces an error by default:

```
open cursor test
commit tran
open cursor test
```

If you set either the **close on endtran** or **chained** options, the cursor's state changes from open to closed after the commit. This allows the cursor to be reopened.

Any exclusive locks acquired by a cursor in a transaction are held until the end of that transaction. This also applies to shared locks when using the **holdlock** keyword, the **at isolation serializable** clause, or the **set isolation level 3** option. However, if you do not set the **close on endtran** option, the cursor remains open past the end of the transaction, and its current page lock remains in effect. It could also continue to acquire locks as it fetches additional rows.

Backup and Recovery of Transactions

Every change to the database, whether it is the result of a single **update** statement or a grouped set of SQL statements, is automatically recorded in the system table *syslogs*. This table is called the **transaction log**.

Some commands that change the database are not logged, such as **truncate table**, bulk copy into a table that has no indexes, **select into**, **writetext** and **dump transaction with no_log**.

The transaction log records **update**, **insert**, or **delete** statements on a moment-to-moment basis. When a transaction begins, a **begin transaction** event is recorded in the log. As each data modification statement is received, it is recorded in the log.

The change is always recorded in the log before any change is made in the database itself. This type of log, called a write-ahead log, ensures that the database can be recovered completely in case of a failure.

Failures can be due to hardware or media problems, system software problems, application software problems, program-directed cancellations of transactions, or user decisions to cancel a transaction.

In case of any of these failures, the transaction log can be played back against a copy of the database restored from a backup made with the **dump** commands.

To recover from a failure, transactions that were in progress but not yet committed at the time of the failure must be undone, because a partial transaction is not an accurate change. Completed transactions must be redone if there is no guarantee that they have been written to the database device.

If there are active, long-running transactions that are not committed when SQL Server fails, undoing the changes may require as much time as the transaction has been running. Such cases include transactions that do not contain a **commit transaction** or **rollback transaction** to match a **begin transaction**. This prevents SQL Server from writing any changes and increases recovery time.

SQL Server's dynamic dump allows the database and transaction log to be backed up while use of the database continues. Make frequent backups of your database transaction log. The more often you back up your data, the less work will be lost if a system failure occurs.

The owner of each database or a user with OPER authorization is responsible for backing up the database and its transaction log with the **dump** commands, though permission to execute them can be transferred to other users. Permission to use the **load** commands, however, defaults to the Database Owner and cannot be transferred.

Once the appropriate **load** commands are issued, SQL Server handles all aspects of the recovery process. SQL Server also automatically

controls the checkpoint interval, which is the point at which all data pages that have been changed are guaranteed to have been written to the database device. Users can force a checkpoint if necessary with the **checkpoint** command.

For more information, see the *SQL Server Reference Manual* and the *System Administration Guide*.

